

AD-A128 692

STATE-OF-THE-ART ASSESSMENT OF TESTING AND TESTABILITY
OF CUSTOM LSI/VLSI..(U) AEROSPACE CORP EL SEGUNDO CA
A J CARLAN OCT 82 TR-0083(3902)04-1-VOL-1 SD-TR-83-20

1/1

UNCLASSIFIED

F04701-80-C-0081

F/G 9/2

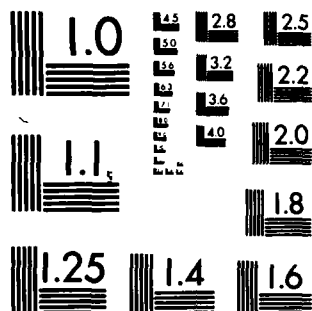
NL

END

DATE

FILED

DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

12

AD A128692

State-of-the-Art Assessment of Testing and Testability of Custom LSI/VLSI Circuits

Volume II: Hardware Design Verification

M. A. BREUER & ASSOCIATES
Encino, Calif. 91436

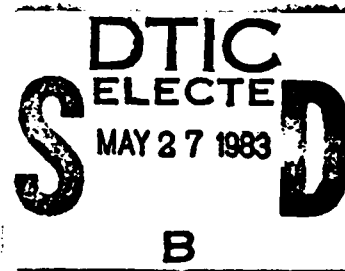
and /

A. J. CARLAN
Technical Study Director

October 1982

Engineering Group
THE AEROSPACE CORPORATION
El Segundo, Calif. 90245

Prepared for
SPACE DIVISION
AIR FORCE SYSTEMS COMMAND
Los Angeles Air Force Station
P.O. Box 92960, Worldway Postal Center
Los Angeles, Calif. 90009



DTIC FILE COPY

APPROVED FOR PUBLIC RELEASE;
DISTRIBUTION UNLIMITED

83 05 26 10 8

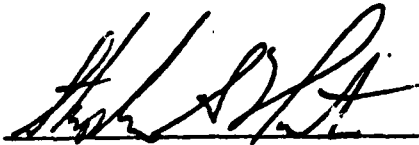
This final report was submitted by the Aerospace Corporation, El Segundo, CA 90245 under Contract No. FO4701-82-C-0083 with the Space Division, Deputy for Logistics and Acquisitions, P.O. Box 92960, Worldway Postal Center, Los Angeles, CA 90009. It was reviewed and approved for The Aerospace Corporation by J. R. Coge, Electronics and Optics Division, Engineering Group. Al Carlan was the project engineer.

This report has been reviewed by the Office of Information and is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nationals.

This technical report has been reviewed and is approved for publication. Publication of this report does not constitute Air Force approval of the report's findings or conclusions. It is published only for the exchange and stimulation of ideas.

FOR THE COMMANDER

APPROVED


STEPHEN A. HUNTER, LT COL, USAF
Director, Speciality Engineering
and Test

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER SD-TR-83-20	2. GOVT ACCESSION NO. AD-A128692	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) State-of-the-Art Assessment of Testing and Testability of Custom LSI/VLSI Circuits Vol II: Hardware Design Verification		5. TYPE OF REPORT & PERIOD COVERED Interim
7. AUTHOR(s) M.A. Breuer & Associates and A.J. Carlan Aerospace Technical Director		6. PERFORMING ORG. REPORT NUMBER TR-0083(3902)04)-1
9. PERFORMING ORGANIZATION NAME AND ADDRESS M.A. Breuer & Associates 16857 Bosque Dr. Encino, CA. 91436		8. CONTRACT OR GRANT NUMBER(s) F04701-80-C-0081 ✓ F04701-81-C-0082 ✓ F04701-82-C-0083 ✓
11. CONTROLLING OFFICE NAME AND ADDRESS Space Division Air Force System Command Los Angeles, Calif. 90009		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) The Aerospace Corporation ✓ El Segundo, Calif. 90245		12. REPORT DATE October 1982
		13. NUMBER OF PAGES 57
		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Hardware Descriptive Languages (HDLs) Program Verification Design Specification Hardware Verification Methods of Specification Petri Nets Design Verification Techniques		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) The complexity of digital circuits requires that more emphasis be placed on design specifications and verification. Specification of design requirements currently advocated is done with formal hardware descriptive languages (HDLs) to describe hardware function. Industry's current use of HDLs is primarily for simulation. Verifying a design is a less mature discipline. Three approaches are considered: simulation, symbolic simulation and formal proofs. While symbolic simulation shows promise, much research and development is required.		

DD FORM 1473
(FACSIMILE)

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

EXECUTIVE SUMMARY

Digital designs have become complex. Specification of design requirements and determination that a design meets these requirements is no longer a casual matter. Correctness of digital designs is the subject of this report.

This report has investigated two major issues:

- How to specify design requirements for digital systems to designers/contractors
- How a designer/contractor can verify or validate that a design meets the above requirements.

Both of these issues have been researched with increasing activity over the past five years. Some results are available and have been applied by industry on a small scale to successfully validate and verify designs.

The specification of design requirements is done with narrative description diagrams, or formal hardware descriptive languages. The most complete technique uses hardware descriptive languages (HDLs) to describe hardware function. An HDL description or program bears strong resemblance in many cases to a software program written to execute the same function on a computer. The main difference between HDLs is the level of detail which can be included. Some HDLs describe interconnected gates; others describe black-box behavior of sequential machines. The more detail which is specified, the more likely a design is to meet requirements. However, supplying large amounts of detail is, in essence, doing the design itself. Thus, there is a balance between supplying too little information, allowing for the possibility of ambiguity and hence design or specification errors, and supplying most of the design itself (which itself may be in error).

83 05 26 10

The state-of-the-art in HDLs has advanced to the point that they are widely used by industry, but primarily for simulation. They are used increasingly for design documentation. The use of such a language for design specification to contractors is practical at this time, with some limitations and exceptions.

Verifying that a design meets some specified requirements is a less mature discipline. There are three approaches to this task

- Simulation
- Symbolic simulation
- Formal proofs

ranked in decreasing order of usage. Simulation is widely used. However, *simulation only proves correctness for the specific cases simulated*. Symbolic simulation, on the other hand, uses symbols as input data to the simulator. The outputs from the simulator are functions of these input symbols. In theory, all possible cases can be examined in this manner. American and Japanese companies have reported results in this area, although it is not known the extent of practical application of this technique.

Formal proofs, much like geometric proofs, for example, have been used in academic studies to prove correctness. This is the most rigorous type of analysis but is not likely to be employed by industry in the near future since it is tedious and only small problems have been examined.

In short, formal languages (HDLs) can be used to specify design requirements. Some validation, via simulation, is possible at this time. Symbolic simulation is a technique which shows promise.

TABLE OF CONTENTS

page

EXECUTIVE SUMMARY	iii
1. INTRODUCTION	1
2. SPECIFICATION OF DIGITAL SYSTEMS	2
2.1 Methods of Specification	4
2.1.1 Graphical Techniques	6
2.1.2 Other Diagrams	10
2.1.3 Hardware Descriptive Languages (HDLs)	16
2.2 Important Features for Specification Techniques	27
2.2.1 Specification of Behavior	27
2.2.2 Specification of Design Rules	28
2.3 Conclusions	29
3. DESIGN VERIFICATION TECHNIQUES	33
3.1 Introduction	33
3.2 Basic Concepts in Program Verification	34
3.2.1 Program Language Definition Techniques	35
3.2.2 Program Verification Scenario	36
3.3 Differences between Program and Hardware Verification	37
3.4 Examples of Hardware Verification and Results	37
3.4.1 Verification of Abstract Behavior	38
3.4.2 Microcode Verification	41
3.4.3 Register-Transfer Level Verification	42
3.4.4 Logic Verification	43
3.5 Design for Verification	44
3.6 The State of Hardware Verification and Conclusions	45
REFERENCES	47

1. INTRODUCTION

This report addresses the problem of logical correctness of digital designs. This problem divides into several subissues:

- How can design requirements be specified unambiguously to a designer?
- How can a designer or design team verify that a design produced from a set of requirements accurately reflect those requirements?
- How can evidence be presented to a contractor that a design does meet the requirements?
- Can certain design techniques support design verification?

We will deal with the first two subissues in detail in this report; the third and fourth are open research questions.

In particular, Chapter 2 deals with design specification. Section 2.1 summarizes informal specification techniques. 2.1.1 reviews graphical techniques, and 2.1.2 covers other diagrams useful for design specification. Hardware-Descriptive Languages (HDLs) are summarized in 2.1.3. Section 2.2 points out some important features of specification techniques, beginning with specification of desired behavior (2.2.1). The specification of design rules to be used in implementing the specified designs is mentioned in 2.2.2. Conclusions about design specification are presented in 2.3.

Chapter 3 deals with the problem of design verification. Section 3.2 discusses various concepts in program verification which are applicable to hardware verification, and in Section 3.3 we discuss the difference between these two problems. In Section 3.5 we assess the field and in Section 3.6 present some ideas on design techniques to aid in verification.

NTIS GRA&I		<input checked="checked" type="checkbox"/>
DTIC TAB		<input type="checkbox"/>
Unannounced		<input type="checkbox"/>
Justification		
By _____		
Distribution/		
Availability Codes		
Dist	Avail and/or Special	
A		



2. SPECIFICATION OF DIGITAL SYSTEMS

This chapter summarizes current techniques for digital system specification. These techniques are discussed in light of the desired levels of description of the digital systems, and the styles of design of the systems being described. Some important features of good specification techniques are presented, followed by a possible scenario for use of HDLs. Although the thrust of this chapter is on requirement specification techniques for contractors to use when processing digital designs, these techniques are equally applicable for documentation of designs in progress.

Before we can address the problem of design specification we need to state and define exactly what information constitutes a design specification. Not all of this information is included in every design specification. Some information may be actually required to meet system requirements external to the chip being specified. Other types of information may overly constrain a designer and therefore produce suboptimal designs. *Thus, there is a balance between ambiguity and precision which allows a designer freedom and still meets requirements, and this balance may shift from design to design.*

There is specific information which must be included in a design specification:

- design requirements
- technological requirements and constraints
- behavior - both internal and external

We will be primarily concerned with the specification of behavior.

Design requirements are requirements which concern the implementation of a design. For example, a design might be specified to be done in the style of a microprocessor (Von Neumann machine) or in the style of a multiple-instruction-single-data stream (pipeline) machine. On a lower level, there might be a requirement that only two-phase clock schemes be implemented, or that all multiplexers be built with transmission gates (design rules).

Technological requirements and constraints involve packaging, power consumption, cost, size, pin count, and any consideration independent of the actual functioning of the IC.

The IC function is termed *behavior*. Behavior has two major components:

- data flow
- control flow

Data flow is the set of functions to be performed on data, along with a specification of input and output variables. The ordering of operations or functions is done on the basis of *data precedence*, which specifies that an operation cannot be performed on some data until the operation which produced that data as output has been performed.

More detail can be added to the data flow specification by indicating:

- actual memory available to the designer
- number of registers required in the design
- number of functional units of each type
(adders, shifters, etc.)
- Interconnections between registers and
functional units
- Bit widths of each element

Control flow is the specification of when and under what circumstances each function specified in the data flow is performed. Precise timing, conditional execution, concurrent operation (greater than one state machine running simultaneously) and *state sequencing* are all part of control flow. State sequencing can be synchronous (clocked) or asynchronous (self-timed). Response to exception (error) conditions is also part of control flow.

Implementation-specific information can be added to the control flow. This includes:

- specification of control hardware in terms of
a finite-state machine
- specification of control hardware in terms of
a microprogram
- assignment of operations to clock phases
- assignment of bit patterns to states of the
state machine or fields of the microinstruction
- specification of communication mechanisms be-
tween concurrent state machines

In summary, the control flow specifies the parallelism to be implemented in the design. This should be distinguished from the data flow, which shows variable interactions and hence the *potential parallelisms*.

Ideally, specification of a design should encompass both control and data flow. Furthermore, it is convenient to divide behavior into internal function and interaction with the external world. The importance of data-flow specification is increased when internal functions are being described; interaction with the environment demands precise specification of control flow.

Several reviews of hardware-descriptive languages are useful for reference [Dudani and Stabler 1978, Shiva 1979] in spelling out the requirements for information contained in hardware descriptions.

2.1 METHODS OF SPECIFICATION

Requirements specifications fall into two classes — specification of the behavior of the design and a set of design rules which should be adhered to in carrying out the design. We address the former in great length; the latter will be touched on briefly.

Specifications of digital system behavior can be informal or formal. Up until the past few years, specification prior to design was virtually always informal.

Informal descriptions are usually just verbal narratives. They are useful as a complement to formal descriptions, which tend to be cryptic. Also, verbal descriptions are used to define formal notations. The main pitfall in relying on informal specifications is that there is an opportunity for ambiguity, or worse, conflicting information, which cannot be detected by computer processing of the description.

It is relevant to note here some research Balzer has performed on the machine processing of informal program specifications [Balzer et al. 1977, Goldman et al 1977]. He has found a number of criteria for assessing whether program specifications are well-formed. For example, if a variable is used in an operation but has not been produced anywhere, then an action has been omitted from the specification. This type of analysis could be used for testing the completeness of informal hardware specifications.

Formal descriptions use well-defined rules to express specific information. This information can be in the form of

- graphs
- diagrams
- languages

The appropriateness of each of these techniques depends on the level of description and style of design. *Level* refers to the degree to which the physical structure of a design is being described. Common terminology and taxonomy usually includes these levels

- logic
- register-transfer
- architecture (behavioral)
- algorithm (system)

Thus, the *logic level* is described by the interconnection of gates and flip flops and the *register-transfer level* by the interconnection of registers and other functional units. The register-transfer level is higher than the logic level since the gate structure of the registers and other units is not included in the higher-level description.

The *architecture level* deals with register transfers, but in an abstract fashion. For example, at this level we might state "put into the A register the contents of memory location 3." We have not specified explicitly the memory-address register, the memory-data register, or the intermediate transfers which must occur. The *algorithm level* abstracts away any information about the hardware, with the possible exception of parallelism (concurrent operations).

The *design style* is the global shape of the data flow, as implemented in the hardware. We can identify four major styles:

- central accumulator
- distributed random
- parallel
- pipelined

Central accumulator design style encompasses designs which process data serially, sharing registers and functional units, and storing intermediate results in random-access memory. Distributed random styles usually contain irregularly connected networks of registers and operators (address, etc.), operating simultaneously. Parallel styles have a regular structure (repeated sub-networks)

but are otherwise similar to *distributed random* styles. Pipelined designs contain an interconnection of operators and registers arranged one after the other, so that a sequence of data values can filter through the "pipe," being passed from register to operator, following the previous value through the pipe. Microprocessors reflect the central-accumulator design style while VLSI layouts of signal-processing algorithms are done in the parallel design style. A summary article on design styles can be found in [Thomas and Siewiorek 1981].

It should be noted here that specification of a single state machine is easier than description of concurrent, communicating state machines, due to the complex control flow inherent in the latter problem.

2.1.1 Graphical Techniques

Graphical techniques for design specification include *state diagrams*, *control-flow graphs*, *Petri nets*, *data-flow graphs*, and *binary decision diagrams*. (The UCLA graph (The Graph Model of Behavior) [Razouk 1977] is a well-known and researched extension of the Petri net.)

State diagrams are a universal technique for system specification, from abstract behavior to the operation of interconnected gates and flip flops. Since they specifically model control flow, they are often used to specify the behavior of a control unit in a digital system. They are also useful in the description of interfaces and protocols for communication between functional units, where control flow is complex. Actions are indicated in the circles (states) and conditions for state change are shown beside the arcs. They have been used for simulation and synthesis of finite-state machines. A portion of the UNIBUS operation described with state diagrams is shown in Figure 1. Note that the actual operations carried out by the hardware can be specified informally (e.g., $npg \leftarrow 1$ means set npg to the value 1), which partially defeats the purpose of the formal specification. Furthermore, timing is not specified formally using this technique. Finally, state diagrams become unwieldy when used for description of large systems (a PDP-8 computer, for example) which cannot be partitioned into multiple, asynchronous state machines. State diagrams were used extensively in the design of the Hewlett-Packard interface bus [Knoblock 1975].

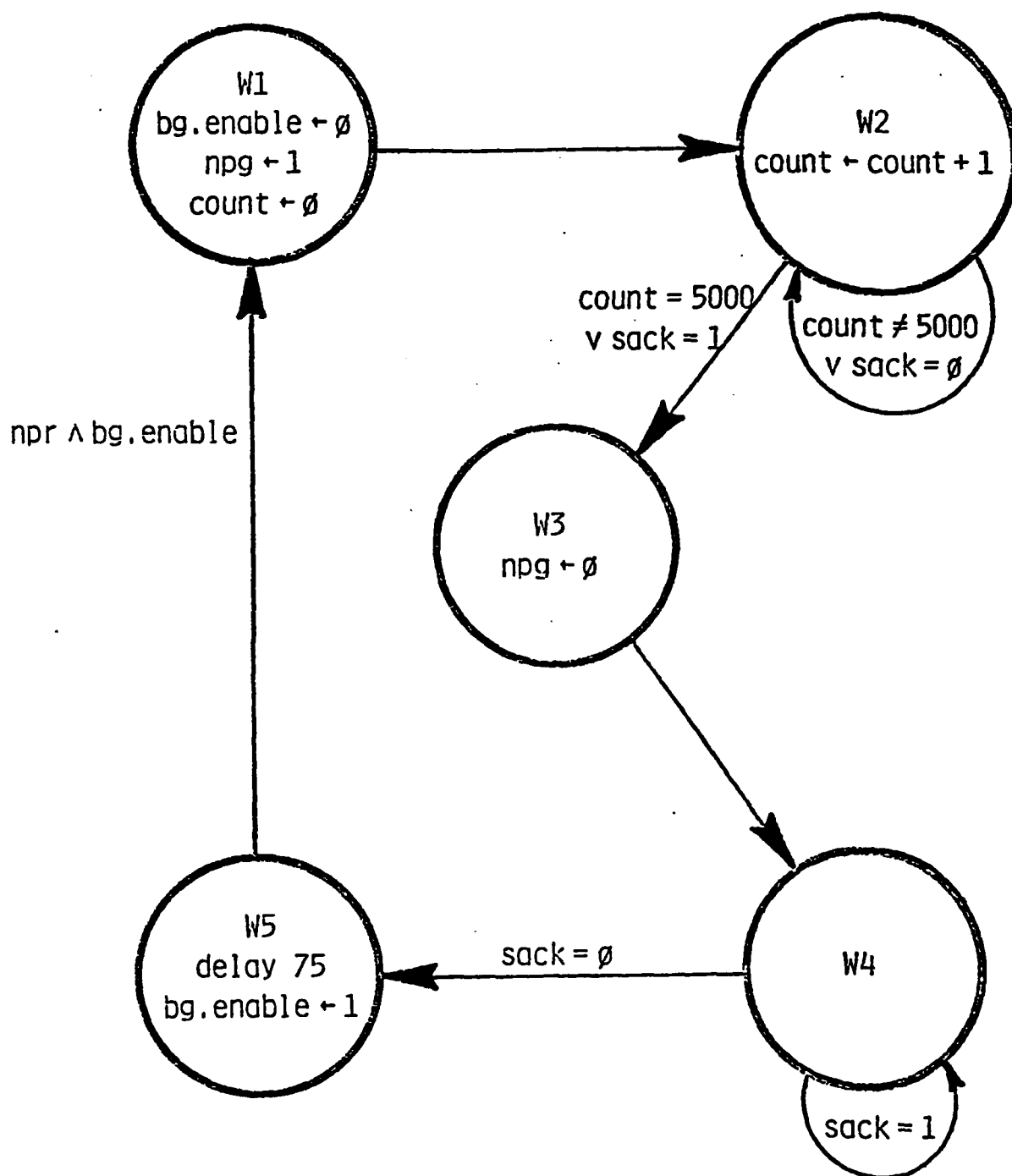


Figure 1: The UNIBUS™ Non-Processor Grant Process,
Described in State Diagram Form. (From
[Parker & Wallace 1981])

Control-flow graphs are a hybrid between flow charts and state diagrams. Figure 2 shows a typical control-flow graph. Each node in this graph specifies a specific operation to be performed and the required ordering of operations is indicated by the directed arcs of the graph. "Fork" and "Join" nodes encapsulate multiple paths in the graph whose operations could occur during the same period of time. Operations to be packed into microinstructions are often presented in this format, as in the Carnegie-Mellon Design Automation (CMU-DA) System [Nagle and Parker 1981]. A control-flow graph for the PDP-11 has been automatically translated into microcode. The advantage of this specification technique is that it can be easily derived from a language description, either manually or by computer. It has the same shortcomings as the state-diagram approach.

Petri nets are used to describe behavior of digital systems, usually above the register-transfer level. Figure 3 shows an example of a Petri net. The use of Petri nets has been primarily for verification of correct behavior and for simulation purposes. Chapter 3 spells out some results of these applications. Petri nets model asynchronous control flow. The circles represent places and the bars represent transitions, or events (register-transfers, for example). When enough tokens accumulate in an input place(s), a transition "fires," a token is taken from every input place and a token is placed on every output place connected to that transition. Petri nets can be processed by algorithms to provide information such as:

- whether the number of states of the machine is finite
- whether there is a directed path from any state to an initial state
- whether it is possible to reach any arbitrary state from any other arbitrary state by a series of firings

The latter two findings tell us whether the machine can be initialized and whether there are deadlocks (states which cannot be left once entered). Processing Petri nets can be computationally expensive, but for some classes of Petri nets this problem can be overcome. Petri nets are useful for asynchronous circuit description and for specification of pipelined operation. Because techniques exist to process Petri nets, they provide a useful specification

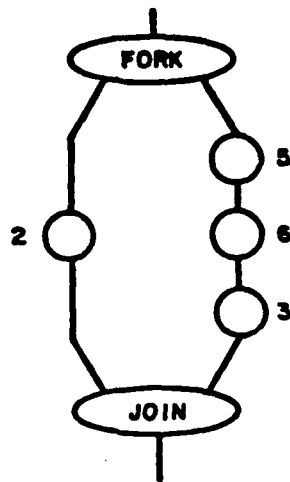


Figure 2. Control Graph of a Single Parallel Segment. (From [Nagle & Parker 1981])

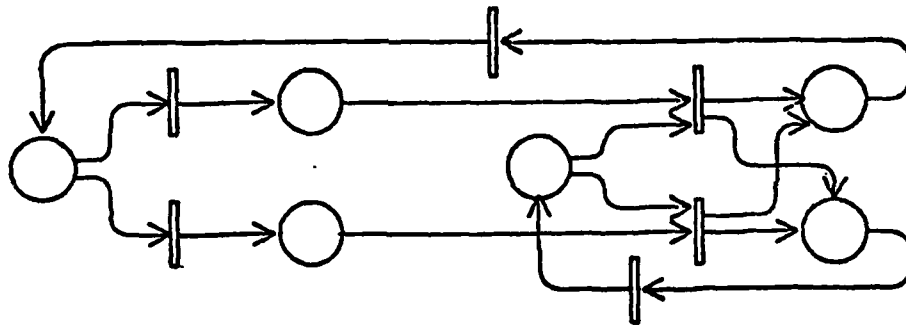


Figure 3. Petri Net Graph.
(From [Cooprider 1976])

format for posing many theoretical problems. Unfortunately, they suffer the same maladies as state diagrams. However, timing has been included in an extension to Petri nets proposed by Wallace [Wallace 1979]. Summaries of Petri nets are found in [Agrawala 1979, Coopridge 1976].

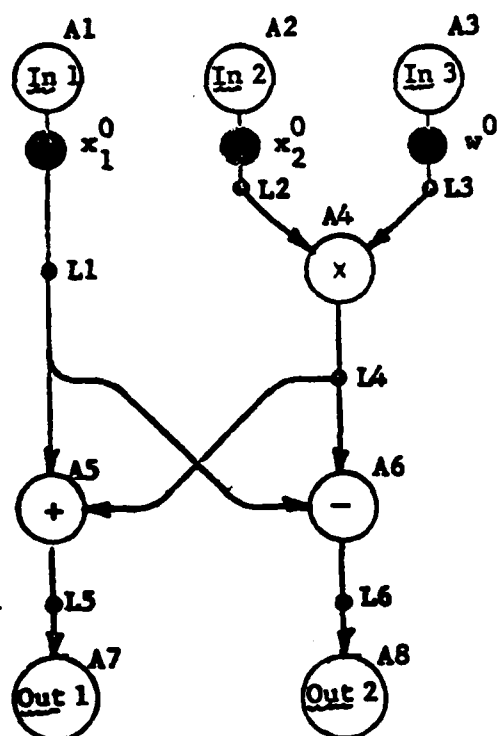
All of the graphs discussed so far are most useful when control flow is important to the designer. When the digital system is to perform a complicated manipulation of data (for example an FFT), the overall design problem may be reflected better with a data-flow graph (Figure 4). The nodes indicate operations performed on values, and the arcs indicate values used or produced. Control flow is implicit, since ordering of operations here depends only on availability of data (data precedence). Data flow graphs generally indicate function or behavior without specifying any hardware structure. These graphs are useful for optimization of the control flow since they remove unnecessary orderings. However, they are not useful for describing precisely-timed behavior or asynchronous communications. Data-flow graphs are primarily used for software optimization or to represent programs to data-flow machine [Dennis et al. 1977]. We should note here that an extension of the data-flow graph, the Value Trace [McFarland 1978, Snow et al. 1978], has been designed. This graph can be automatically derived from a language description of hardware behavior, and plays a part in some verification research described in Chapter 3.

Binary decision diagrams [Akers 1980] (Figure 5) specifically model the logical behavior of combinational and sequential logic. No timing considerations are included. It is possible that binary decision diagrams or an extension of these diagrams could be used more easily than conventional logic diagrams for VLSI design, since transmission-gate structures could be modeled. They have been used for verification of logical behavior.

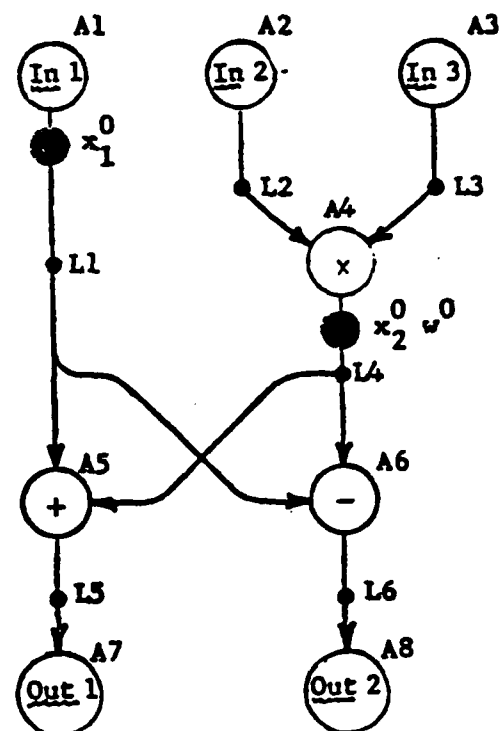
2.2.2 Other Diagrams

There are other types of diagrams useful for system specification. The first of these is the *system block diagram*, or PMS diagram [Bell and Newell 1971], which usually contains data interconnections between independent functional units; control connections are often omitted since they tend to be implementation dependent. More dynamic information can be provided by hardware

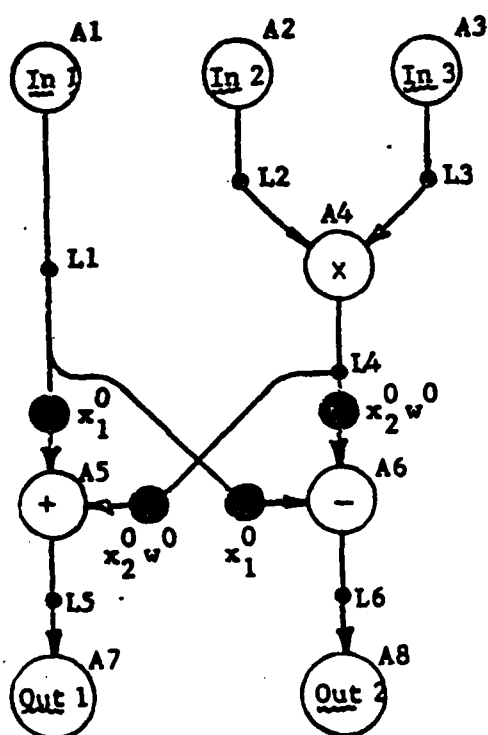
(a)



(b)



(c)



(d)

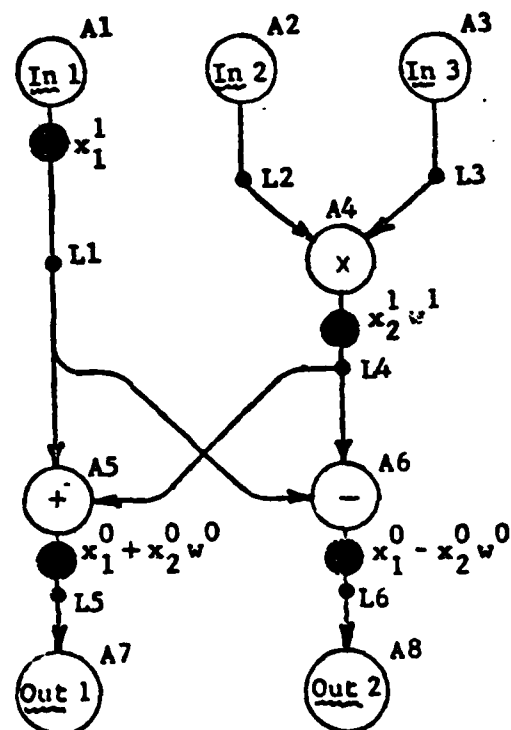


Figure 4. Snapshots of a Data Flow Program in Execution.
(from [Dennis et al. 1977])

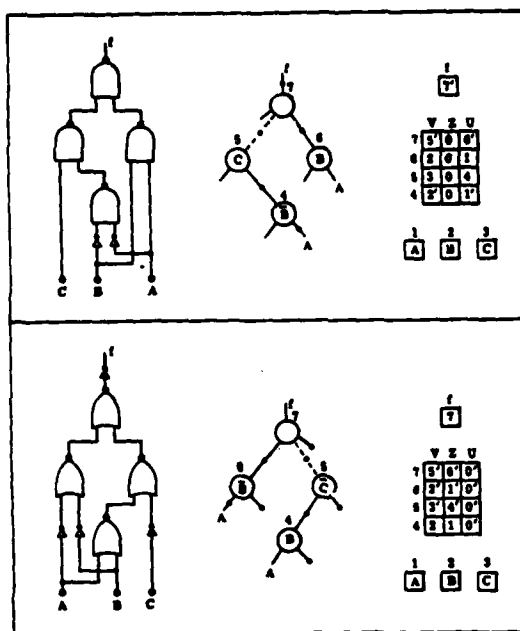


Figure 5. Binary Decision Diagram (from [Akers 1980]).

flowcharts [Tredinnick 1980]. These flowcharts were used by Tredennick to design the control hardware for Motorola's 16-bit microprocessor, the MC68000. These flowcharts assume the register-transfer structure of the data paths has already been designed, and specify the control flow.

The dimensional flowcharts proposed by Lawson [Lawson 1979] are similar to conventional flowcharts but tend to represent abstract behavior in a structured manner. They do not represent timing constraints. A self-explanatory example of Lawson's method is shown in Figure 6. These flowcharts are valuable primarily at the algorithm level of abstraction, and it is not known whether they have had practical application.

Another flowchart approach is a graphic language proposed by Bayegan [Bayegan et al 1979]. Register transfers and conditional signals are shown

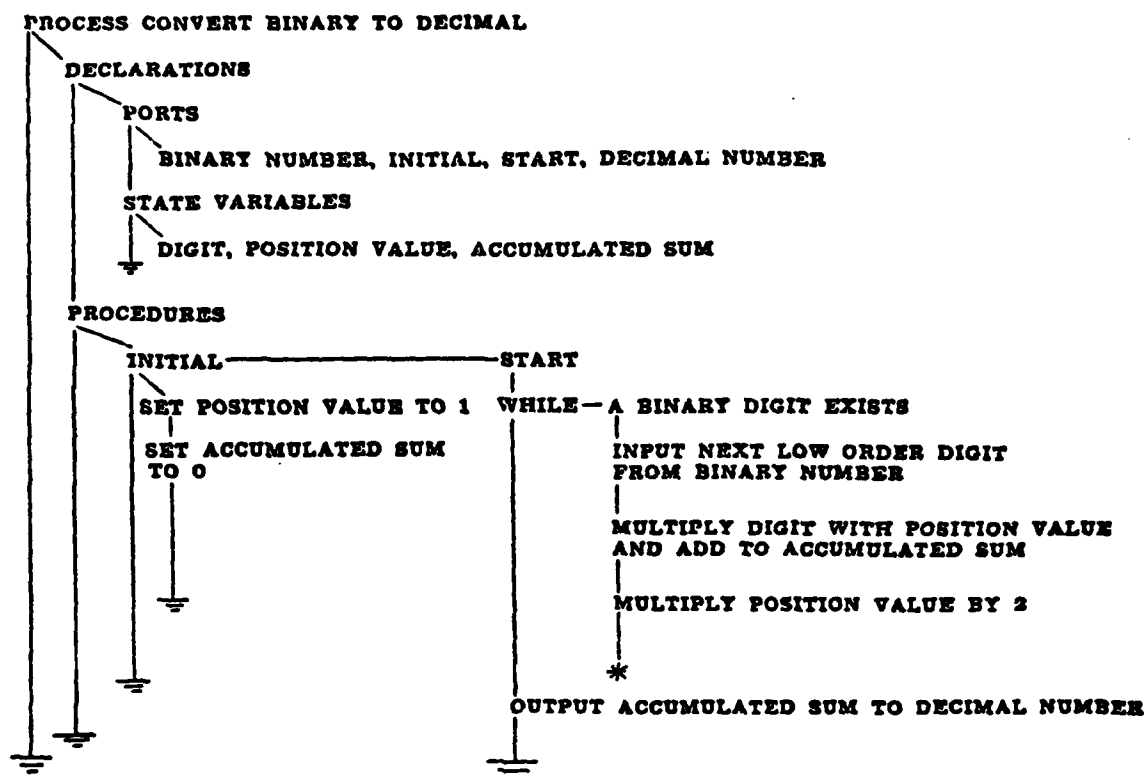
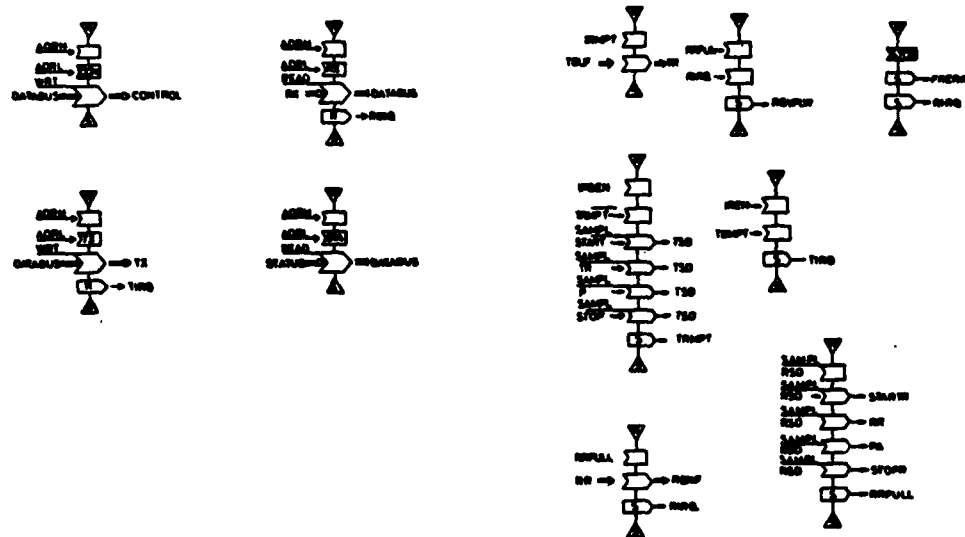


Figure 6. (from [Lawson 1980]).

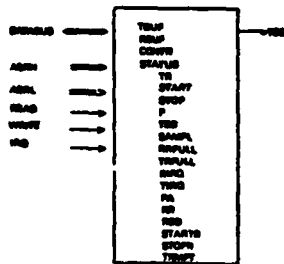
as horizontal arrows and control flow as vertical arrows. (See Figure 7 for some example graphs.) This technique is useful for hierarchical description of hardware and should be investigated more completely.

A final flowchart approach is the ASM (Algorithmic State Machine) chart described in [Clare 1973]. These charts specify control flow of state-machines at the logic level and can be used to synthesize logic designs.

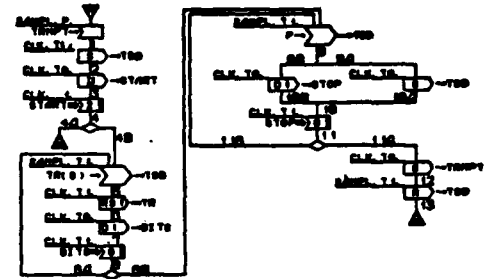
Timing diagrams are the final type of descriptive technique we will touch on in this section. These diagrams are useful particularly when describing how a device interacts with the external world, as shown in Figure 8. They are used extensively in the UNIBUS description [DEC 1979]. Although timing diagrams can be used to describe required operation of hardware at any level,



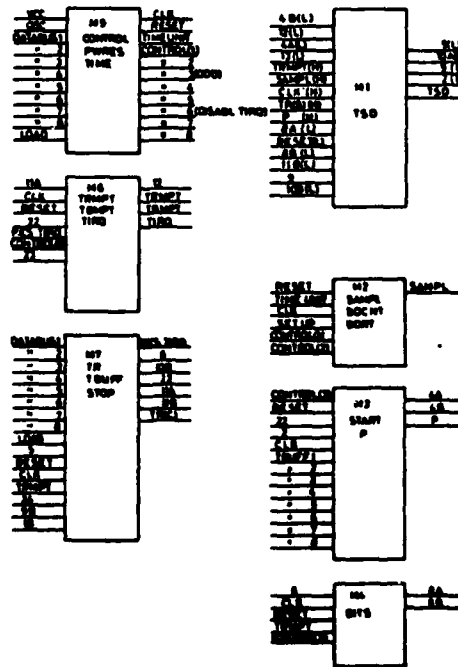
(a) The Behavioral Description of the UART at Top Level



(b) Top Level Functional Macro for the UART.



(c) Part of the Behavioral Description of the Transmitter.



(d) The Macro Elements Belonging to Figure 7c.

Figure 7. From [Bayegan et al. 1979].

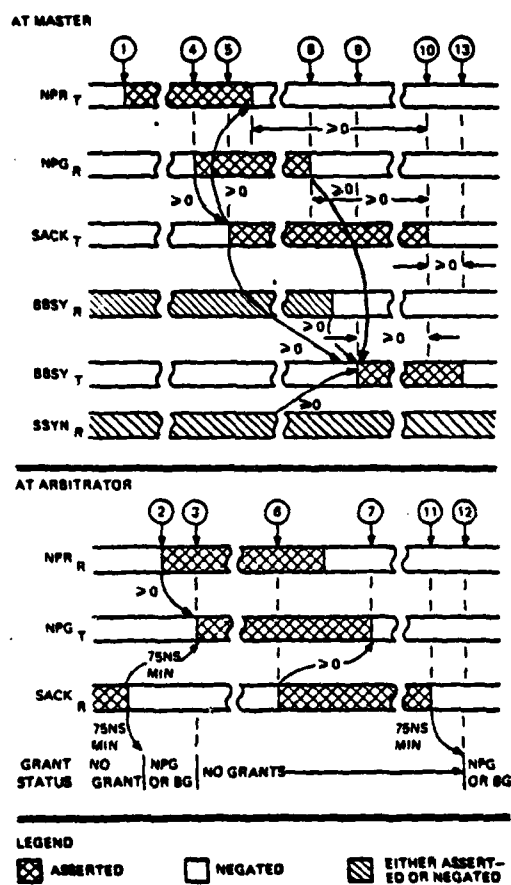


Figure 8. Timing of the UNIBUS[™] Non-Processor Grant Process (from [DEC 1979]).

they are most often used to describe register-transfer or logic behavior. They primarily represent the timing of control flow and do not describe data manipulations.

2.1.3 Hardware Descriptive Languages (HDLs)

Hardware descriptive languages do not fall into the clean categories which the above techniques did. In this report it is impractical to survey the entire field of hardware descriptive languages; [CHDL 1979] provides a selection of current publications. Here, we focus instead on desirable features of languages and on example languages which are popular or which have interesting features which are uncommon.

Two excellent comparisons of hardware descriptive languages (HDLs) (which were published some time ago [Barbacci 1975, Figueroa 1973] revealed some desirable properties of general-purpose HDLs. Barbacci cited

- Readability
- Familiarity with naming and usage conventions
- Use across several levels of detail
- Simplicity; small number of language primitives
- Extensibility
- Fidelity to the system organization
- Timing and concurrency
- Syntactic simplicity
- Ability to separate data and control

These were chosen presumably because of his emphasis on the description of digital systems. Barbacci defined *timing* and *concurrency* as "parallel actions," and explained "At the RT level, concurrent activities are described by allowing them to be activated simultaneously (i.e., under the same conditions)." Figueroa, focusing on design automation, also found some of the above properties important; and in addition discussed

- Modularity
- Block structuring, including the existence of global and local variables
- Facilities for functions, subroutines and macros
- Facilities for specification of parallel processes
- Facilities for determining and controlling process interaction explicitly.

In addition, some comments on programming languages are valid here as well. Iverson [Iverson 1980] finds valuable *the ability to subordinate detail and economy* — the ability to express a larger number of ideas in terms of a relatively small vocabulary. In addition he recommends that the languages be *amenable to formal proofs* and *suggestive* — that is forms of expressions suggest related expressions found in similar problems. Winograd [Winograd 1979] considers the overall structure of programs. He proposes that

- states and state transitions be expressed
- modularity and structured procedures be employed
- interacting processes and their communication be specified

Other features of good programming languages which are relevant here are [Higman 1967]

- the use of a standard character set
- the use of established constructs in the field of the problem being specified
- the prohibition of unintentional ambiguities

Thus, we will judge HDLs not merely by their expressive capabilities but also with respect to the features previously mentioned here.

Example languages can be subdivided into those which describe a design at the level of

- system or algorithm
- architecture (behavior)
- register transfer
- logic

However, some languages cover a mixture of these levels depending on the individual language constructs. Others apply equally well to more than one level.

System level behavior can be described with a programming language such as PASCAL or ADA. Sequential orderings in the language, however, should not constrain the hardware to the same sequential control flow. The SCHEME chip [Holloway 1980] was described and designed in a top-down fashion using the LISP language. Since the SCHEME chip is a processor which executes LISP, this

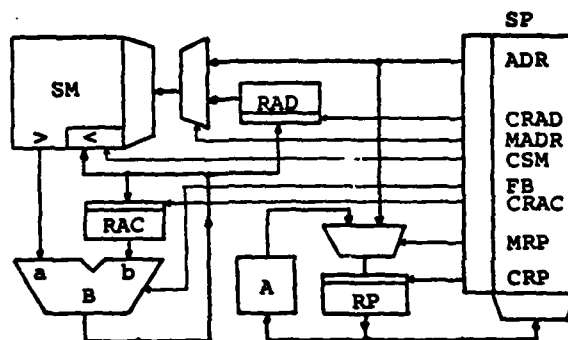
allowed a close correlation between specification technique and desired system behavior.

An example of a system-level language is MIMOLA [Zimmerman 1979]. It has been used successfully as input to an automatic design system which produces an architecture and machine organization as output. In one example, MIMOLA was used to describe the behavior of an operating system, and the architecture produced was designed to support such a system. The advantage of MIMOLA is that it does not constrain the designer even to a specific architecture. However, MIMOLA lacks the ability to express multiple-process communication easily and cannot provide timing constraints. Finally, the syntax of MIMOLA makes the language difficult to read. An example of structural description using MIMOLA is shown in Figure 9. Figure 9(a) shows the hardware and 9(b) shows the language description. Behavior is indicated with register-transfers, and control flow by ALGOL-like constructs. (IF-THEN-ELSE, WHILE-DO, etc.).

Another publication in this area [Sorenson 1978] describes a system level language based on the BCPL language. This language has three interesting features - the ability to declare processes (concurrency), mailboxes for inter-processes communication (synchronization), and uninterruptable code sequences (critical sections). The language is designed to model systems at a higher function level than register-transfer behavior and consists of a mixture of block diagrams (each block represents a concurrent process) and statements (which represent process behavior).

Another system-level language along similar lines, SOL (Simulation Oriented Language) [Knuth 1964], based on ALGOL-60, does not describe hardware structures explicitly. However, it does allow concurrent processes, timing, subordinate processes and shared-resource contention to be dealt with explicitly. One other system-level language, ASPOL [MacDougall 1973] also reflects these important aspects of interconnection behavior. Concurrent processes can be declared, and processes can have priorities of execution. Set and Wait primitives are provided for synchronization. While these languages provide synchronization and process-level constructs, they do not provide the level of detail possessed by register-transfer languages.

Usually, some structure (or architecture) of a system has been determined during this initial design process. In this case, a language can express *register-transfer behavior or architecture* is required.



Simple Processor Example

(a)

```

DECLARE
  ADDMODUL
    SM<A(16384:0).BIT(15:0), SM>A
      "main memory, 16k, 16 bits, 2 ports",
    SP(1024:0).BIT(21:0) "microprogr.mem",
    RAC.BIT(9:0) "accumulator",
    RAD.BIT(15:0) "address register",
    RP.BIT(9:0) "program counter",
    A(.INCR) "monadic operator",
    B(+,-,.a,.b) "adder",
    I.BIT(21)CRP.BIT(20)MRP.BIT(19:18)FB
      .BIT(17)CRAC.BIT(16)CSM.BIT(15)MADR
      .BIT(14)CRAD.BIT(13:0)ADR
      "microprogram word, field names";

```

Modul Declaration

(b)

Figure 9. MIMOLA Example (from [Zimmerman 1979]).

There is strong motivation for constructing *behavioral* register-transfer (RT) descriptions of hardware. Behavioral RT descriptions differ from structural RT descriptions because they describe only the RT functions of the hardware and not the RT hardware itself. Some storage locations and register-transfers which exist in the hardware may be absent from the behavioral description; control hardware is implicit rather than explicit.

Behavioral RT descriptions can convey to the reader the overall operation of hardware better than structural RT descriptions, since much of the unnecessary detail is eliminated. Simulations proceed more rapidly, and can encompass larger systems. Verification of behavior is possible since the behavior is explicit, the implementation is hidden, and the description can be structured.

Of course, more detail is present in architectural and behavioral RT language than is the case with system-level languages. Bit widths are assigned to variables, and arithmetic operations are often expressed in terms of more primitive functions. The control flow contains specific sequencing and concurrency information. Most descriptions at the behavioral register-transfer level are *procedural* or *operational*. Control flows sequentially from statement to statement unless otherwise altered. (There are specific exceptions to this when behavior becomes *non-procedural*. A particular condition can trigger a set of actions at any time, for example.)

ISPS [Barbacci 1979] is an example of such a behavioral RT language. The applications of ISPS are numerous. They include

- simulation
- machine description for automatic compiler generation
- automatic synthesis of digital hardware
- architectural evaluation
- automatic generation of microcode

An example ISPS description is shown in Figure 10. ISPS has shortcomings, some of which are discussed in [Parker et al. 1979]. In summary, these deal with lack of formal definition of the language semantics, and the presence or absence of language features which have made machine processing of ISPS descriptions difficult.

One particular shortcoming of ISPS is its inability to deal with the concurrent processes which characterize system-level specifications. Thus, ISPS is most useful describing isolated-state-machine situations.

The Mark-1 Computer, [Lavington, 1975]

```

Mark1 :=
  Begin
    ** Primary.Memory **
    M[0:8191]<31:0>,
    ** Central.Processor **
    Processor :=
      Begin
        ** Processor.State **
        PI\Present.Instruction<15:0>,
          F\Function<0:2> := PI<15:13>,
          S<0:12> := PI<12:0>,
        CR\Control.Register<12:0>,
        Acc\Accumulator<31:0>,
        ** Instruction.Cycle **
        I.Cycle :=
          Begin
            PI = M[CR]<15:0> next
            Decode F =>
              Begin
                0\JMP := CR = M[S],
                1\JRP := CR = CR + M[S],
                2\LDN := Acc = - M[S],
                3\STO := M[S] = Acc,
                4:5\SUB := Acc = Acc - M[S],
                6\CMP := If Acc Lss 0 =>
                  CR = CR + 1,
                7\STP := Stop(),
              End next
            CR = CR + 1 next
            Restart I.Cycle
          End
      End
  End

```

Primary Memory Section
 8K words, 32 bits/word

Central Processor Section

Processor State Section
Instruction Register
 Opcode
 Operand Address
 Program Counter

Instruction Interpretation Section

Instruction Fetch
Instruction Decoding

Jump Indirect
Jump Relative
Load Complement
Store
Subtract
Conditional Skip

Halt

Increment Program Counter
Repeat Instruction Cycle

The MARK-1 Computer

Figure 10. Example ISPS Description (from [Barbacci 1979]).

SLIDE [Parker 1978, Parker and Wallace 1981, Altman and Parker 1980] has been designed for description of concurrent processes at the register-transfer level. At this point SLIDE has all the features of ISPS with the exception of procedure parameters, the case construct, and some arithmetic and logic operations. On the other hand, many SLIDE features are not present in ISPS.* The major shortcomings of ISPS with respect to interconnected system description are at two extremes — the lowest level hardware operations and the overall process structuring. For example, attempts to describe the UNIBUS in IPS have resulted in inability to synchronize between concurrent processes and failure to model the open-collector behavior, timing, and transitions of signals on the bus itself. SLIDE has some non-procedural constructs which allow more accurate modeling. A portion of the UNIBUS described in SLIDE is shown in Figure 11.

```

busgrant + / NEXT    ! grant the bus !
DELAY 500 UNTIL ack EQL /
    ELSE BEGIN      ! do this if a timeout !
        sysreset + / NEXT    ! reset the bus !
        DELAY 100 NEXT
        sysreset + \
    END

```

Delay statement example with timeout

Figure 11. SLIDE Example (from [Wallace & Parker 1979]).

A specific construct lacking at this level is the ability to replicate interconnection structures between hardware modules algorithmically. For example, a set of processors and memories is connected through a crosspoint

*The most recent version of ISPS has incorporated some SLIDE features for process synchronization.

switch. Each element in the switch must be specified individually in order to describe the system. Sastry [Sastry 1980] has proposed a language construct which allows algorithmic replication of interconnection structures.

The SLIDE and ISPS research produced a list of requirements for the ideal behavioral register-transfer language. Here is the list, taken from [Parker et al. 1979]:

1. An abstraction facility for adding new primitives to the language* as hardware becomes more complex.
2. The ability to specify behavior without specifying structure.
3. Support for structured programming constructs.
4. The capability to specify additional information which may be application specific (e.g., with qualifiers).
5. The capability to express concurrency more precisely than the ISPS semicolon now allows.
6. The capability to describe multi-process functionality, including terminations, suspensions, and priorities.
7. The capability to express synchronization primitives explicitly.
8. Formal semantic definition of the language operators to the greatest possible extent.

ADLIB [D. Hill 1979] is a language which attempts to span the gap between system-level languages and behavioral RT languages. It is built onto PASCAL, and thus has some semantic definition as well as software support. A disadvantage of ADLIB is the syntax, which becomes unwieldy as the level of description becomes lower. A multiplexer description, for example, requires many (> 20) lines of ADLIB code. There are some interesting timing features and concurrent operation allowed at the process level (independent executing environments). However, concurrent operations can be described only by declaration and use of an explicit synchronizing signal. ADLIB comes the closest to possession of description capabilities across hardware levels of the languages surveyed. An ADLIB example of combinational logic is shown in Figure 12. It should be noted, however, that the power of ADLIB is in description of higher levels of behavior.

* These primitives may not be functions of existing primitives, so simple text substitution is not all that is implied here.

```

combinator combin;
input
a,b,c : boolnet;
output
d : boolnet;
internal
x : boolnet;
subprocess
endgate : transmit (a and b) to x
         delay 15.0;
orgate : transmit (x or c) to d delay 14.0;
begin
permit(endgate);
permit(orgate);
end;

```

Combinational logic:
 $D = (A * B) + C$

Figure 12. ADLIB Example (from [Hill and Van Cleemput 1979]).

DDL [Dietmeyer 1978] is an example of a *structured register-transfer language*; as such it conveys more detail than ISPS and the underlying semantics are more straightforward since operations are more primitive. Control flow is explicit and unstructured (not structured in the sense that there are "go-to" conditions, just as in unstructured software). A DDL example is given in Figure 13.

Since there is more detail than in a comparable ISPS description, DDL descriptions quickly become large. Furthermore, since these descriptions are essentially unstructured they do not reflect multiple-process operation clearly. DDL is a *nonprocedural* or *declarative* language. Each statement in the language executes whenever the conditions for its execution become true. DDL has been used widely for simulation and synthesis of gate-level designs, as well as verification.

AHPL is similar in structure to DDL. AHPL III [Hill 1979] does allow replication of interconnections. An example using AHPL III is shown in Figure 14.

Shiva reported AHPL and DDL were both difficult to program with [Shiva 1979].

IX	Addressing Mode	Effect of Executing an ADD Instruction
00	Immediate	$ACC \leftarrow ACC + XADR$
01	Direct	$ACC \leftarrow ACC + M[ADR]$
10	Relative	$ACC \leftarrow ACC + M[B + ADR]$
11	Indirect	$ACC \leftarrow ACC + M[M[ADR]]$

$\langle ID \rangle XADR = ADR[1] \times 6 \cdot ADR, ZADR = 0D6 \cdot ADR.$

IF1 : $MAR \leftarrow CAR, \rightarrow IF2.$

IF2 : $IR \leftarrow M[MAR], CAR \leftarrow CAR + 1, \rightarrow IF3.$

IF3 : $[OP] \text{ add } [IX] \{0\} ACC \leftarrow ACC + XADR, \rightarrow IF1$

[1] $MAR \leftarrow ZADR, \rightarrow EX$

[2] $MAR \leftarrow ZADR + B, \rightarrow EX$

[3] $MAR \leftarrow ZADR, \rightarrow IF4...$

IF4 : $MAR \leftarrow M[MAR], \rightarrow EX.$

EX : $ACC \leftarrow ACC + M[MAR], \rightarrow IF1.$

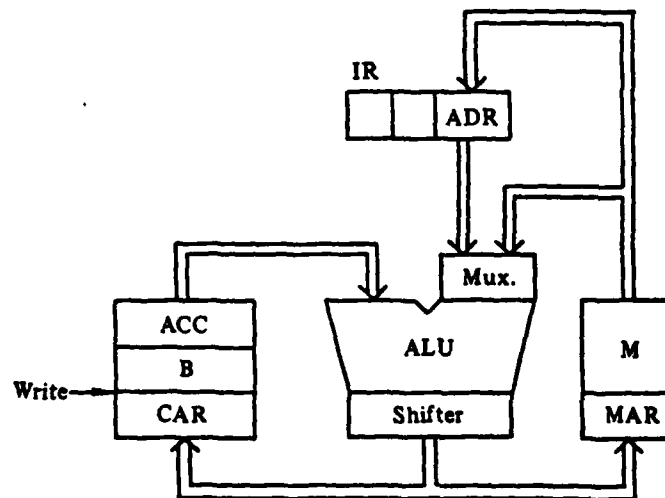


Figure 13. DDL Example

Combinational logic unit description may be written in this structured syntax as illustrated by a 16-bit ripple carry adder.

```

UNIT: ADD(X,Y)
:
:
begin body C[16] = 0
FOR I = 15 DOWNTO 0 CONSTRUCT
C[I] = CARRY_I(X[I],Y[I],C[I+1])
S[I] = SUM_I(X[I],Y[I],C[I+1]) ROW
ADD = C[0],S[0:15]
end body.

```

(a)

```

FUNCTREG: COUNTER(load, enp, ent, clear; DATA)
begin declarations
DATA INPUTS: DATA [4].
CONTROL INPUTS: load, enp, ent, clear.
OUTPUTS: COUNTER [5].
MEMORY: D[4]
CLUNITS: INC[4]
end declarations
begin

$$D^*((enp \wedge ent) \vee \overline{load}) + (INC(D) \wedge DATA)$$


$$\quad \quad \quad \wedge ((enp \wedge ent), \overline{load})$$


$$D^{\overline{clear}} + 0, 0, 0, 0;$$

COUNTER =  $ent \wedge (\wedge/D), D$ 
end

```

(b)

Figure 14. (a) AHPLIII Adder Example
(b) AHPLIII Counter Example

We conclude this section with a discussion of CONLAN (consensus language) being designed by a committee [CONLAN 1979]. CONLAN in essence consists of a base language and mechanisms for building more specific languages onto the base language. The main advantage of CONLAN seems to be the use of the base language to *define* any other hardware-descriptive language. There are some desirable features of CONLAN. These include

- distinguishing physical interconnections from the action of a register-transfer.
- algorithmic replication of functional units.
- the provision for insertion of assertions about environment conditions into the CONLAN descriptions. These assertions indicate what is supposed to be true at the point they are asserted.

Even so, there are some constructs present in current hardware descriptive languages which represent semantics (meaning) not easily specified in CONLAN.

In summary, the features of a language for hardware description (and the applicability of a given language) depend on each application. It is more difficult to choose a good language when the range of applications and number of levels of design (e.g., logic and register transfer) increases.

2.2 IMPORTANT FEATURES FOR SPECIFICATION TECHNIQUES

We have already presented desirable features of hardware descriptive languages. Here, we focus on the use of languages and graphs for *specification of design requirements*. As such, we subdivide these into specification of behavior, and specification of required design rules, or design practice.

2.2.1 Specification of Behavior

We are going to emphasize here important features of specification techniques which are not common to current methods. It is assumed that basic features (like the ability to express sequential behavior and conditional actions) are required features.

The specification of internal device (functional unit) functioning is easier than specification of external behavior. The main areas of ambiguity are usually

- arithmetic expressions
- exception (error) conditions

Accuracy of arithmetic computations is an important consideration if designs are specified above the register-transfer level. At the RT levels the exact meaning of arithmetic operators should be made clear.

The most commonly omitted aspect of design specification is the precise description of response to error or unanticipated conditions. For example, the use of the PDP-11 condition codes to detect exceptions is not clearly defined or uniformly employed [Russell 1978]. This is an area which requires research.

External behavior must be specified more precisely than internal function in order to provide a uniform interface between independently produced modules. Just as in system specification, the concept of process communication must be present. Sequencing and timing of events which appear at the interfaces to the modules must be specified, including allowable worst-case and desirable response times. Also, crucial timing relationships between signals which are not allowed to change over time should be indicated. Finally, the behavior of the module in producing or processing streams of signals, particularly synchronous transfers, should be clearly specified.

2.2.2 Specification of Design Rules

More general than the specification of a particular design is the specification of design rules to be followed during implementation.

Design rules are quite often expressed informally. For example, at the register-transfer level, a commonly followed design rule is the following: In order to store a value, it must have been available at the register inputs for at least the set-up time of the register prior to the clock.

A more desirable method of expression is, of course, a formal technique. Such a technique would allow unambiguous description and would indicate clearly where conflicts between design rules existed.* A formal expression of the above rule is: The storage element s_1 is a clocked register with input set-up time $D_{SS}(s_1)$. For the store of the value 0_1 in register s_1 , the clock time must satisfy the constraint

* It would also allow automatic checking for design rule violations.

$$T_{OS}(O_1) \cong T_{OA}(O_1) + D_{SS}(s_1)$$

where $T_{OS}(O_1)$ is the time s_1 is clocked to store value O_1 and $T_{OA}(O_1)$ is the time the outputs of operation x_1 are available for use. (Data path delay is not being modeled.)

A summary of design rules for the register-transfer level is found in [Hafer and Parker 1981], along with their application to automatic design. Their role in design verification is presented in Chapter 3. Extension of these design rules to cover other levels and aspects of design is an area of future research.* McWilliams [McWilliams 1980] uses some specific timing rules which will be discussed in Chapter 3.

2.3 CONCLUSIONS

Ideally, a single specification technique should suffice for specifying design requirements and documenting the design process. This technique should provide clear specification of

- communicating and computing hardware processes
- multiple levels of designs
- detailed interface behavior along with less-detailed internal functioning
- asynchronous and synchronous operation

The technique should support

- modularity and structured description
- ease of use of mathematical tools for analysis
- static checks for inconsistencies, rule violations, and ambiguities

Along with the technique, a formal semantic definition of the technique should be provided so that the precise meaning of the language is understood.

Along with this specification technique, and, interspersed with specific descriptions, design rules should also be expressed in a uniform formal manner.

* Rules for design for testability at the RT level are currently being researched.

Finally, it should be clear from the specification which aspects are merely expressions which allow the description to be written, and which are requirements. For example, the use of a temporary register T_1 may be necessary in order to describe a complex computation, but may not be needed in the actual hardware. A second temporary register T_2 , on the other hand, may be available externally and hence must be realized.

In reality, the above techniques are not currently available. No one specification method is adequate. A possible scenario is to use a number of techniques, some redundant, in expressing design requirements. (In fact, redundancy may be desirable here to ensure understanding since the specification techniques will not be ideal.) An example of combined techniques is the use of delay information combined with state diagrams for protocol description [Visser 1978].

The scheme we mention here is the following: Designs should be specified in a hardware-descriptive language at the behavioral (architectural) register-transfer level. Gate-level specifications will not be appropriate for VLSI and VHSI designs since transistor structures now can be used to directly implement register-transfer functions, and these structures do not reflect logic diagram specifications of the same function. If register-transfer structure is required to be the level which the contractor specified designs, DDL should be used since it has had wide usage and is relatively readable. However, it should be noted that the specification of register-transfer structure constrains the designer to a control flow and interconnection topology, thus predetermining floor plan, speed and cost to a large extent. If DDL is used, timing diagrams should be used in conjunction to express interface behavior, since deducing timing relationships between signals in a DDL description is difficult. State diagrams or Petri nets should be used to indicate overall behavior, and carefully correlated with the detailed description.

One approach is that systems should be specified internally at a more abstract level than the structural RT level, while maintaining physical interconnections and timing requirements in the interface specification. SLIDE does allow this, and could be used independently of any other technique. Informal definition of SLIDE control-flow constructs has been done using Petri-net-like graphs. However, SLIDE data manipulations have never been precisely defined. For this reason, an alternate approach is to use ISPS for internal specification.

A formal definition of ISPS does exist and is expressed in a notation which could be rewritten for use by vendors. (The current notation is straightforward but compact and hence cryptic.) [McFarland 1980].

In summary, areas for future study include:

- The use of graphic languages like that proposed by [Bayegan 1979].
- The feasibility of combining elements of ISPS, SLIDE, ADLIB and DDL to produce an ideal language.
- The formal definition of the selected specification language semantics (meaning).
- The precise specification of exception conditions in the language.
- The specification of design rules for all designs done from contractor specifications.
- The ability to separate design *requirements* from abstract expressions of behavior in a specification.

3. DESIGN VERIFICATION TECHNIQUES

3.1 INTRODUCTION

One of the fundamental problems facing VLSI designers is that of assuring themselves that an integrated circuit is functionally correct. By functionally correct, we mean that the outputs and their ordering are what we expected from our inputs. In other words, there are no logical errors in the design. This is a difficult problem with LSI circuits, but an overwhelming one with VLSI and VHSI designs. The complexity of the designs and the timing considerations are difficult to keep track of. There are two approaches to determining the functional correctness of hardware.

- Simulation
- Verification

Simulation does not guarantee that a design is logically correct unless all input data combinations have been applied during the simulation. It merely validates* that for the inputs chosen, the correct outputs are obtained, and in the correct order.

Verification, on the other hand, gives some confidence that a design is correct for an entire class of inputs, or for all possible inputs. Simulation handles specific cases, while verification involves the general case. An analogy can be given: Suppose we suspected that the three angles of a triangle added up to 180. We could construct a large number of cases which all gave us the expected result. Or, we could prove, using geometric theorems and axioms, that for all triangles, our rule held. Verification of hardware involves the same type of analysis in proving that a particular hardware design meets certain rules we pose about its behavior.

Simulation of hardware as a validation technique is widely used. We will not pursue this topic further here. Rather, we intend to survey the field of hardware verification. Within the timeframe of this study, only a brief glimpse of the field could be taken - many more aspects of the field should be investigated. [Cory and VanCleemput 1980] recently published a survey paper on this subject.

Section 3.2 presents some basic concepts of program verification which are also applicable to hardware verification. Section 3.3 describes the differences between program verification and hardware verification. The variety

*Validation of correctness is an informal or case by case method for establishing correctness.

of hardware verification techniques which have been published are summarized in 3.4. 3.5 presents some ideas for design for verification, and 3.6 gives and assessment of the field.

Verification is a difficult, complex research field. All that can be done here is to present the concepts in as straightforward manner as possible. The reader should try to view verification merely as the manipulation of strings of symbols in order to produce equivalent strings. Whether the symbols represent program code or hardware descriptions, that is all that is really going on. Definitions and axioms merely tell us how to produce and manipulate these strings.

Another way of viewing verification is to view it as *functional testing* where faults are not in the fabrication of the design, but in the design itself.

3.2 BASIC CONCEPTS IN PROGRAM VERIFICATION

The fundamental basis of program verification is the concept of proof of equivalence. We may have either of the following situations:

- Two descriptions of a program at the same level of detail which we want to prove equivalent.
- A detailed description (a program) and an abstract description (a specification) which we use to specify the required behavior of the program.

In either case we need an underlying *definition* of each of the elements of the language or specification technique so that we can manipulate the descriptions until we can prove equivalence. In other words, we need to understand exactly what each statement in the program language does. We also need to understand what we mean by each operation in the specification if we give an abstract specification. Then we need some axioms, or rules, or methodology to manipulate the detailed description (or one of the detailed descriptions if two are to be compared), until it is identical to the other description.

Often, in program verification, the program is compared to the abstract specification. The abstract specification is expressed as a set of assertions about the program behavior. These assertions can be further specialized into *preconditions* and *postconditions*. In essence, preconditions give the conditions existing prior to execution of a particular program statement or segment;

postconditions give us the desired conditions afterwards, given the preconditions which existed beforehand. Alternately, we can state that, given postconditions which exist after execution, the preconditions are what we would have expected to exist beforehand.

3.2.1 Program Language Definition Techniques

As we said earlier, we could not prove two descriptions of a desired computation equivalent unless each aspect of both descriptions was carefully defined. Research in program verification has paid much attention to the ability to define the exact meaning of various operations in a given programming language. The underlying model of program behavior can be one of two type :

- Computational
- Sequential

Computational behavior of each statement in a programming language is simply the specification of the mapping of inputs to outputs. Thus, it is concerned with the effect of each program statement on the flow of data (data values).

Sequential behavior, on the other hand, is concerned with a series of actions, some of which may involve the environment. A sequential model of behavior of a program statement or segment must therefore express the series of actions which occur as a result of the statement execution. Most program verification has been involved with computational behavior specification and definition. In other words, PROGRAMMERS ARE MAINLY CONCERNED WITH GETTING THE RIGHT OUTPUTS FROM THE POSSIBLE INPUTS TO THE PROGRAM. THEY ARE GENERALLY NOT CONCERNED WITH THE SEQUENCING OF ACTIONS IN ORDER TO GET THE OUTPUTS. We shall return to this difference in a few paragraphs.

Here, we merely summarize definition techniques which have been used. The summary is mainly abstracted from [McFarland 1981*]. The earliest model of computational behavior definition techniques was the use of the interpreter function. This function was used to specify the mapping from inputs to outputs for each action in a program. For example, the FORTRAN statement

$$A = B+C$$

would have as its underlying definition a function which took two operands as

* Michael McFarland is a Ph.D. student at Carnegie-Mellon working under the supervision of Alice Parker.

input and produced a single operand as output (which was the sum of the two inputs), and would be specified as the mapping from the lattice of inputs to the lattice of the outputs. (A lattice is an ordered set.) The interpreter function formed the basis for other models of behavior which defined program actions. These are

- operational definitions
- denotational semantics [Gordon 1979]
- propositional semantics

An operational definition gives rules for translation of programs into instructions for an abstract, predefined machine. Denotational semantics defines each program element as a function over a domain. Thus, an assignment statement would be defined by a function which operated on the domain of possible data structures. Thus, the *denotation*, or meaning, of a program is a function which maps each initial state into a final state. Propositional semantics requires preconditions (what is known to be true before execution) and postconditions (what must be true after execution if the precondition held prior to execution). Thus a definition of a program element would be the postconditions which would result after execution, if the preconditions existed prior to execution.

3.2.2 Program Verification Scenario

Using one of the above definition techniques, we could set out to prove a program correct using one of two techniques:

- symbolic execution [King 1969]
- formal proofs

Symbolic execution has a lot of intuitive appeal. In symbolic execution we execute a program, only instead of using numerical inputs, we use symbols. We retain these symbols throughout the execution of the program, and obtain expressions for the outputs in terms of the input symbols. If these output expressions can be then manipulated so that they are equivalent to the high-level specifications, or equivalent to the output from the symbolic execution of the comparison program, then the program has been proved correct. Usually, the high-level specifications are in terms of preconditions and postconditions. The inputs are symbols which adhere to the preconditions, and the output expressions are compared to the postconditions.

The expressions involved in symbolic execution tend to get unwieldy with long programs, particularly when there are lots of branches. If a program is instead translated into the lower level definition, then that definition can be mathematically manipulated, much as in a geometric proof, until it is shown to be equivalent to another program or specification, defined in the same manner. Thus, formal techniques for proving equivalence by algebraic manipulations can also be used. Operational definitions and denotational semantics are often used to define programs which are proven equivalent in this manner.

3.3 DIFFERENCES BETWEEN PROGRAM AND HARDWARE VERIFICATION

Verifying hardware is more difficult than verifying software because, in the general case, both the control flow and data flow must be verified. Hardware often does not reach a final state. It cannot be described as a mapping from inputs to outputs. Rather, hardware behavior is better described as the sequence of outputs obtained from a sequence of inputs, interleaved in a prescribed manner. In specific cases, the focus has been on either control flow or data flow. Either the control flow or the data flow is considered the problem, and the other is assumed correct or insignificant.

If we are concerned with the ordering of actions, we must deal with the sequential definitions of behavior mentioned earlier. These include state diagrams (or more generally, finite state machines), Petri nets, path expressions [Habermann 1974], and other sequential models. A few hybrid models do take both control flow and data flow into account. These include UCLA graphs [Razouk 1977], extensions of denotational semantics, and transition systems [Keller 1976].

Thus the main difference between program verification and hardware verification is the desire to verify orderings of events as well as the events themselves.

3.4 EXAMPLES OF HARDWARE VERIFICATION AND RESULTS

We are going to subdivide the examples of hardware verification into the following areas:

- abstract behavior
- microcode
- register-transfer-level hardware
- logic-level hardware

We present examples in each area and the outcome of each study. Levels of verification below the logic level, such as design rule checks for layout, will not be dealt with.

3.4.1 Verification of Abstract Behavior

Abstract behavior is behavior that is at the architectural or functional register-transfer level. Verification at this level usually consists of

- protocol verification
- functional verification

Protocol verification is concerned with the communication mechanism between two or more modules in a system. Verification involves insuring that there are no deadlocks, or that no module is continually denied access to the communication mechanism. Finite state machines and Petri nets are commonly used for protocol verification. The following discussion of these is from [McFarland 1981].

If the system has a significant amount of data memory which must be modeled, the fsm formalism becomes cumbersome. There must be a separate state assigned to each possible set of values in the memory, which makes consideration of the set of all possible state transitions impossible. In modeling a simple sender-buffer-receiver communications protocol, for example, a finite state machine analysis can be used to verify that the three processes cycle correctly through their control states ("send," "wait," "receive," "acknowledge," "error"). But in a more complicated protocol, where the actions being taken depend in a complicated way on the content of the message being passed, a separate state would have to be assigned to each set of values for the messages currently in the system, making the usual fsm analysis techniques to unwieldy.

The Petri net model of system behavior is similar in some ways to the finite state machine model, but is more general, in that it can also describe the concurrent execution of several processes.

Because it is not required to have a single locus of control, i.e., a single enabled state as in an fsm, a Petri net may be used to model a system composed of several concurrent processes and their interfaces.

In practice Petri nets, like finite state machines, are very effective in modeling overall flow of control, but do not show the details of the data state or the data transformations. If the control flow depends in a critical way on the particulars of the data being processed, then Petri nets by themselves are not adequate for the analysis.

Both finite state machines and Petri nets present the dynamics of a system in terms of lists of transition rules or firing rules. This information is relatively unstructured and diffuse. It is hard to get any sense of how the system works from looking at the individual rules themselves. It is only the composition of the rules into sequences of transitions and the patterns of states resulting from them that are of interest. Some sense of the patterns can be derived from graphical representations; but any meaningful, rigorous analysis usually involves "simulating" in some way the allowed sequences of transitions until all possible sequences have been explored. The lack of structure also makes these models more difficult to represent in a logical system and to reason about, although some work in the hierarchical decomposition of Petri nets has been done.

UCLA graphs have also been used for protocol verification [Razouk and Estrin 1980]. McFarland also discusses UCLA graphs:

Another system based on a graphical control structure similar to a Petri net with information added to describe data transformations is the UCLA Graph Model of Behavior (GMB). A system is described in the GMB model by a *data graph*, a *control graph* and an *interpretation*. The data graph shows the data path structure of the system, i.e., registers and memories, processors and their interconnections. The control graph is similar in concept to a Petri net in that it models control flow by passing tokens among nodes, but has a somewhat different structure. There is only one type of node in the graph, called a control node. The control nodes are associated with processors in the data graph, so that when a control node is activated, its associated processor performs its function. A control node is activated when the proper combination of tokens are present on its input arcs. After its process has completed, the activated node removes the tokens from its input arcs and places a certain combination of tokens on its output arcs, guided possibly by conditions on the data state. The nodes in the control graph, therefore, are roughly analogous to the transitions of a Petri net, and the arcs to places.

The interpretation specifies the formatting of the data in the memory elements and the specific functions of the processors. It is through the data graph and interpretation that the GMB can model data transformations as well as control flow.

Van-Mierop [Van-Mierop 1979] has used the GMB, along with the SARA system of hierarchical design specification, to model a class of concurrent processes. Within this model he has shown how to verify the "proper cycling" of a system, meaning essentially that the system always returns to its main control loop. The method he used was to generate all "interesting" states via symbolic execution and to check that every such state was reachable from every other one, so that there were no deadlocks or subloops from which escape was impossible.

McFarland covers a final study [Brand and Joyner 1978]:

In this application a nondeterministic program consisting of several parallel processes is checked against a simple single-process machine which describes the required sequential behavior for the protocol. Again the designer must supply stopping points and a correspondence relation. Both machines are "executed" symbolically and corresponding stopping points compared in order to verify that the correspondence relation holds. For the parallel program it must also be shown that at every stopping point all processes but one are waiting or delayed. Since the stopping points are usually the places where shared data is accessed or changed, if only one process is active at each such point, there is no chance of conflict.

Transition systems have also been used for protocol verification [Bochmann 1975] and [Bochmann and Gesei 1977]. Other research has been reported, but was not read in the scope of this study [Hailpern and Owicki 1980].

All of the above techniques involve an analysis that looks at all cases, in some fashion. Formal proofs have also been used to verify protocols, but this work is in the early stages.

Progress in the protocol verification field has been good. Quite a few real-world protocols have been analyzed. A good summary of the field is found in [Sunshine 1979].

Unfortunately, most hardware verification problems relevant to VLSI fall into the less developed field of functional verification. Here, the communication between modules is deemphasized, and the behavior of modules themselves is explored.

The most recent work in this area is the thesis by McFarland [McFarland 1981]. McFarland begins with the ISPS hardware description language. His goal is to prove that manipulations of ISPS descriptions to optimize the cost or speed of the resultant implementations do not change the fundamental behavior of the machine being described. He first defines every element of the ISPS language. He also invents a way to specify the fundamental behavior desired of the machine at the level more abstract than the ISPS description. This behavior involves sequences of actions as well as data inputs/outputs. He then derives the fundamental behaviors of the machines which are to be proven equivalent from their ISPS descriptions. Finally, he proves the fundamental behavior equivalent using formal proofs. The first step, that of

defining the ISPS language elements, was done so that he could formally derive the fundamental behaviors from the language descriptions of the machine.

The reader should not become discouraged over the complexity of McFarland's work. It is an early attempt in this area and should be viewed as basic research at this time. He has proved manually that many optimizations performed on ISPS descriptions do produce equivalent behavior.

In an earlier piece of work, Alfvin published a denotational semantics definition of AMDL, which is a hardware descriptive language with roots in ISPS [Alfvin 1979]. The use of this definition for verification has not been published to date. In another effort, Oakley [Oakley 1979] performed symbolic execution of ISPS descriptions, primarily to obtain formal descriptions of the effects of instruction execution, as described in ISPS. This information could then be used in automating the design of compilers, since there, it is necessary to know the effects of executing a particular machine instruction.

3.4.2 Microcode Verification

Because microprograms are quite similar to software programs, verification of microcode can be accomplished using some of the same techniques. Again, McFarland has summarized this field [McFarland 1981]:

Another use of symbolic execution is in microcode verification. This involves proving that a host machine executing a given microcode program does the same thing as the target machine which it is supposed to be emulating. This application is somewhat different in that two "programs" or hardware descriptions are being compared to one another rather than a program being verified against some abstract specification expressed by pre- and postconditions. In existing microcode verification systems, the person verifying the microcode must enter in a *correspondence relation* between the host and target machines, showing how the registers of the target are mapped onto those of the host. The user must also choose corresponding "stopping points" in the two descriptions, i.e., control points at which the states of the two machines must be compared. Usually it is required that there be at least enough stopping points to break every loop in both machines, so that only loop-free segments must be checked. Given this input from the user, the system symbolically executed the two machine descriptions along their respective paths between corresponding stopping points. For each such pair of paths, the system checks that at the end of the paths the required correspondence relationship holds, meaning that corresponding registers hold the same symbolic values. This method has been used in the IBM microcode verification system described by [Birman 1974 and Joyner, Carter and Leeman 1976]. A similar approach, but making use of State-Deltas, has been taken by [Crocker, Marcus and van Mierop 1979]. The State-Delta system adds another level of

verification to that of host-target correspondence in that it can also be used to verify the function of the target machine against an abstract specification, using standard program verification techniques.

Lewinski [Lewinski 1980] demonstrates three methods for verifying microprogram correctness. Besides symbolic simulation, he also uses relational algebra and assertions to do formal proofs of correctness. His work is mathematical and still in the research stage.

3.4.3 Register-Transfer Level Verification

At the register-transfer level, verification of function becomes of immediate interest to industry. Currently, software which simulates at the register-transfer structural level is widely employed. However, the complexity of even present LSI designs disallows exhaustive functional testing of designs. An early look at verifying register-transfer level designs was done by [Hoehne and Piloty 1975]. Hoehne and Piloty make several points worth considering. First, they describe errors which can occur at the RT design level as

- syntactic
- semantic
- timing

errors. Syntactic errors are those which can be detected directly from the RT design, for example, attempting to store an 8 bit value in a 4 bit register. Timing errors can be checked from the RT design alone, also. Semantic errors, on the other hand, indicate differences between the specification and implementation.

They further discuss types of timing errors which occur and give some rules for checking for these errors. Then they describe a method for semantic checking. A set of states are selected so that there is a correspondence between hardware and functional specification. Then, both specification and realization are simulated and corresponding states are checked for equivalence.

One successful research project is the LCD project at IBM, described in [Cory and Van Cleemput 1980].

Wagner [Wagner 1977] performed formal proofs in order to verify simple register-transfer level hardware. He described the hardware using a simple register-transfer language and used symbolic manipulation to determine if the hardware satisfied some specification. His technique was quite powerful. For example, he could detect races, hazards, and oscillations in hardware descriptions, as well as static hazards in conditional expressions. However, he could not deal with flip-flop designs and delays biased to insure correct operation, and he could not model capacitive storage.

The problem with Wagner's research is the size of descriptions he processed. He verified a synchronous counter, a binary ripple counter, and a multiplier.

In a more recent publication [Maruyama et al. 1980], a small processor was verified at the register-transfer level. DDL was used to describe the processor. Backward symbolic simulation (execution) was used, starting with the desired outcome, and working backward through the DDL description until a contradiction to the expected result could be obtained. At times, the human had to intervene to guide the backward tracing.

Denotational semantics has been used for formal definition of register-transfer language elements, and for use in automatic production of hardware simulators [Meinen 1979]. It has also been used for similar definitions for automatic microcode productions [Hansen and Leszczylowski 1980]. Neither of these publications indicates that the definitions were used for verification, however.

A somewhat different approach has been taken by McWilliams [McWilliams 1980] in his research. He used simulation to verify the timing behavior of register-transfer circuits, and included assertions (postconditions) in the simulation input to verify the behavior against.

3.4.4 Logic Verification

Early work in hardware verification was done by Roth [Roth 1977]. This method involved a high-level specification of combinational logic, and its automatic translation to a gate-level logic description. The gate-level description is then compared to a manual design. The assumption made is that the automatically-generated design is correct. The comparison is carried out by a D-algorithm type process. It starts with outputs from the combinational logic, and assigns opposite values to the pairs of outputs under

consideration. By working backward to the inputs, the algorithm looks for inputs which provide a counter-example to the logic equivalence [Roth 1977].

Akers has done similar work [Akers 1980]. He converts logic diagrams to binary decision diagrams and then compares two of the diagrams for equivalence. He manipulates the diagrams until they are equivalent. Both of these techniques deal with combinational logic only.

Leinwand [Leinwand 1979] has produced a method of deriving register-transfer function from sequential logic circuits in order to verify that they exhibit the proper register-transfer behavior. This method was used to recognize counters and adders from their logic circuits.

Darringer [Darringer 1979] describes an approach to sequential logic verification using symbolic execution. The example he gives is a two-bit ripple counter. However, this example is not indicative of the limit on the size of problem capable of being analyzed by this procedure.

Cory and VanCleemput [Cory and VanCleemput 1980] also describes work done at Fujitsu on gate-level verification.

3.5 DESIGN FOR VERIFICATION

One idea that has promise is the notion that designs should be constructed so that their verification is straightforward. This is similar to the design for testability idea. In fact, designs which are easily tested for fabrication faults are also probably easily tested for logical errors. Some possible approaches to this problem are:

- permit only synchronous design
- force control to be microcoded
- require the state be controllable and observable to the outside
- use a hierarchy of design descriptions and maintain correspondence between them
- require the designer to meet assertions given in the design specification
- partition designs
- use only timing-independent design techniques where possible

- use automatic design programs which have been tested for correctness. Double check by verifying human designs against the automatically generated ones.

3.6 THE STATE OF HARDWARE VERIFICATION AND CONCLUSIONS

In summary, formal proofs and symbolic simulation have both been used for hardware verification. Symbolic simulation is more intuitive and perhaps of more immediate value to industry.* From the activity apparent in the field recently (note that there are very few references to hardware verification prior to 1977) it is clear that this is a new research area. However, the potential for usefulness is high, the references are not entirely academic, and the work is beginning to be applied to production designs.

Perhaps significant is the use of human intervention in automatic verification techniques. Verification, much like design, should be directed by humans with the creative decisions under manual control, but with the painstaking details and bookkeeping under control of a machine.

*For example, microcode verification using symbolic simulation has been of practical use [Carter 1977].

REFERENCES

- Agrawala, T., "Putting Petri Nets to Work," *Computer*, vol. 12, no. 12, December 1979, pp. 85-94.
- Akers, S., "A Procedure for Functional Design Verification," *Proc. 10th Int'l. Symposium on Fault-Tolerant Computing*, Kyoto, Japan, October 1980, pp. 65-67.
- Alfvén, P.W., "A Formal Definition of AMDL," USC-ISI Report RR-79-79, Information Sciences Institute, Marina del Rey, California, October 1979.
- Altman, A. and Parker, A., "The SLIDE Simulator: A Facility for the Design and Analysis of Computer Interconnections," *Proc. 17th Design Automation Conf.*, Minneapolis, June 1980, pp. 148-155.
- Balzer, R., Goldman, N., and Wile, D., "On the Use of Programming Knowledge to Understand Informal Process Description," USC-ISI Report RR-77-63, Information Sciences Institute, Marina del Rey, California, October 1977.
- Barbacci, M., "A Comparison of Register Transfer Languages for Describing Computers and Digital Systems," *IEEE Trans. on Computers*, vol. C-24, no. 2, February 1975, pp. 137-150.
- Barbacci, M., "Instruction Set Processor Specifications (ISPS): The Notation and its Application," Carnegie-Mellon University, Computer Science Technical Report CMU-CS-79-123, May 1979.
- Barbacci, M., Barnes, G., Cattell, R., and Siewiorek, D., "The Symbolic Manipulation of Computer Descriptions: The ISPS Computer Description Language," Carnegie-Mellon University, Computer Science Technical Report, August 1979.
- Bayegan, H., Baadavik, O., and Kirkaune, O., "An Interactive Graphic High Level Language for Hardware Design," *Proc. Int'l. Symposium on Computer Hardware Description Languages*, Palo Alto, California, October 1979, pp. 184-190.
- Bell, C. and Newell, A., *Computer Structures: Readings and Examples*, McGraw-Hill Book Company, New York, 1971.
- Birmann, A., "On Proving Correctness of Microprograms," *IBM Journal of Research and Development*, vol. 18, no. 4, May 1974.
- Bochmann, G.V., "Logical Verification and Implementation of Protocols," *Proc. IEEE 4th Data Communications Symposium*, 1975.
- Bochmann, G.V. and Gecsei, J., "A Unified Model for the Specification and Verification of Protocols," *Proc. IFIP Congress*, 1977.
- Brand, D. and Joyner, W.H., "Verification of Protocols using Symbolic Simulation," *Computer Networks*, vol. 2, 1978.
- Carter, W.C., Ellozy, H.A., Joyner, W.H., and Leeman, G.B., Jr., "Techniques for Microprogram Validation," IBM Research Report RC 6361, IBM, Yorktown Heights, New York, January 1977.

RESEARCH AND DEVELOPMENT

- CHDL, *Proc. 4th Int'l. Symposium on Computer Hardware Description Languages*, Palo Alto, California, October 1979.
- Clare, C., *Designing Logic Systems Using State Machines*, McGraw-Hill Book Company, New York, 1973.
- Conlan, ., unpublished presentations, 4th Int'l. Symposium on Computer Hardware Description Languages, Palo Alto, California, October 1979.
- Cooprider, L., "Petri Nets and the Representation of Standard Synchronizations," Carnegie-Mellon University, Computer Science Technical Report, January 1976.
- Cory, W.E. and Van Cleemput, W.M., "Developments in Verification of Design Correctness: A Tutorial," *Proc. 17th Design Automation Conference*, June 1980, pp. 156-164.
- Crocker, S.D., Marcus, L., and van-Mierop, D., "Microcode Verification Project: Final Report," USC-ISI Technical Report WP-17, Information Science Institute, Marina del Rey, California, December 1979.
- Curtis, D., IDS, An Interface Description Systems, unpublished note, ALCOA.
- Darringer, J.A., "The Application of Program Verification Techniques to Hardware Verification," *Proc. 16th Design Automation Conference*, June 1979, pp. 375-381.
- DEC Staff, *PDP11 Bus Handbook*, Digital Equipment Corporation, Maynard, Mass., 1979.
- Dennis, J., Misunas, D., and Leung, C., "A Highly Parallel Processor using a Data Flow Machine Language," MIT Laboratory for Computer Science, Computation Structures Group Memo 134, January 1977.
- Dietmeyer, D., *Logic Design of Digital Systems*, Allyn and Bacon, New York, 1978.
- Dudani, S. and Stabler, E.P., "A Survey of Hardware Description Languages," Syracuse University - Rome Air Development Center, 1978.
- Figuerola, M.A., "Analyses of Languages for the Design of Digital Computers," Technical Report, Coordinated Science Laboratory, University of Illinois, Urbana, Illinois, May 1973.
- Goldman, N., Balzer, R., and Wile, D., "The Inference of Domain Structure from Informal Process Descriptions," USC-ISI Report RR-77-64, Information Sciences Institute, Marina del Rey, California, October 1977.
- Gordon, M., *The Denotational Description of Programming Languages: An Introduction*, Springer-Verlag, New York, 1979.
- Haberman, A.N., "Path Expressions," Carnegie-Mellon University, Computer Science Technical Report, 1974.

- Hafer, L. and Parker, A., "A Formal Method for the Specification, Analysis and Design of Register-Transfer Level Digital Logic," to be presented at the 18th Design Automation Conference, Nashville, June 1981.
- Hailpern, B.T. and Owicki, S.S., "Verifying Network Protocols using Temporal Logic," Technical Report No. 192, Computer Systems Laboratory, Stanford University, June 1980.
- Hansen, I. and Leszczylowski, J., "On Fundamentals of Computer-Aided Design of Firmware," *Proc. 13th Microprogramming Workshop*, November 1980, pp. 3-12.
- G.R. Hellestrand, "MODAL: A System for Digital Hardware Description," *Proc. 4th Int'l. Symposium on Computer Hardware Description Languages*, Palo Alto, California, October 1979, pp. 131-137.
- Higman, B., *A Comparative Study of Programming Languages*, American Elsevier Publishing Company, Inc., New York, 1967.
- Hill, D., "ADLIB: A Modular, Strongly-Typed Computer Design Language," *Proc. 4th Int'l. Symposium on Computer Hardware Description Languages*, Palo Alto, California, October 1979, pp. 75-81.
- Hill, D. and vanCleemput, W., "SABLE: A Tool for Generating Structured, Multi-Level Simulations," *Proc. IEEE Design Automation Conference*, 1979 (IEEE and ACM).
- Hill, F. and Navabi, Z., "Extending Second Generation Software to Accommodate AHPL III," *Proc. 4th Int'l. Symposium on Computer Hardware Description Languages*, Palo Alto, California, October 1979, pp. 47-53.
- Hoehne, H. and Piloty, R., "Design Verification at the Register-Transfer Language Level," *IEEE Trans. on Computers*, vol. C-24, no. 9, September 1975, pp. 861-867.
- Hoffman, M.G., "Hardware Implementation of Communication Protocols: A Formal Approach," *Proc. 7th Symposium on Computer Architecture* (IRISA and ACM Sigarch), May 1980.
- Holloway, J. et al., "The SCHEME-79 Chip," MIT AI Memo No. 559, January 1980.
- Iverson, K.E., "Notation as a Tool of Thought," *Comm. ACM*, vol. 23, no. 8, August 1980, pp. 444-465.
- Johnson, W.A., Crowley, J.J., and Ray, J.D., "Mixed-Level Simulation from a Hierarchical CHDL," *SIGDA Newsletter*, vol. 10, no. 1, January 1980, pp. 2-10.
- Joyner, W.H., Carter, W.C., and Leeman, G.B., "Automated Proofs of Microprogram Correctness," Ninth Annual Workshop on Microprogramming, IEEE, September 1976.
- Keller, R.M., "Formal Verification of Parallel Programs," *Comm. ACM*, vol. 19, no. 7, July 1976.

- King, J.C., "A Program Verifier," Ph.D. Thesis, Electrical Engineering Department, Carnegie-Mellon University, September 1969.
- Knoblock, D., Loughry, D., and Vissers, C., "Insight into Interfacing," *IEEE Spectrum*, vol. 12, no. 5, May 1975, pp. 50-57.
- Knuth, D. and McNeley, J., "A Formal Definition of SOL," *IEEE Trans. Computers*, vol. C-13, August 1964, pp. 409-414.
- Lawson, H., Jr., *Understanding Computer Systems*, Lawson Publishing Company, Linköping, Sweden, 1979. ISBN 91-7372-333-9.
- Lawson, H., Jr., "An Approach to Improving Computer Literacy," Linköping University, Linköping, Sweden, unpublished manuscript, 1980.
- Leinwand, S. and Lamdan, T., "Design Verification Based on Functional Abstraction," *Proc. 16th Design Automation Conference*, San Diego, California, June 1979, pp. 353-359.
- Lewinski, A., "System for Symbolic Hardware Verification," *Proc. 10th Int'l. Symposium on Fault-Tolerant Computing*, Kyoto, Japan, October 1980, pp. 68-71.
- MacDougall, M.H., "The Simulation Language SIML/1," *Proc. National Computer Conference*, 1979, pp. 39-44.
- MacDougall, M. and McAlpine, J., "Computer System Simulation with ASPOL," *Proc. Symposium on the Simulation of Computer Systems*, September 1973, pp.
- Maruyama, F. et al., "Hardware Verification and Design Error Diagnosis," *Proc. 10th Int'l. Symposium on Fault-Tolerant Computing*, Kyoto, Japan, October 1980, pp. 59-64.
- McFarland, M., "The Value Trace: A Data Base for Automated Digital Design," Master's Thesis, Department of Electrical Engineering, Carnegie-Mellon University, Pittsburgh, Pennsylvania, December 1978.
- McFarland, M., "A Formal Definition of ISPS for Proving Properties of Hardware Descriptions, unpublished report, Department of Electrical Engineering, Carnegie-Mellon University, April 1980.
- McFarland, M., "Mathematical Models for Formal Verification in a Design Automation System, Ph.D. Thesis (in preparation) Department of Electrical Engineering, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1981.
- McWilliams, T., "Verification of Timing Constraints on Large Digital Systems," *Proc. 17th Design Automation Conference*, Minneapolis, June 1980, pp. 139-147.
- Meinen, P., "Formal Semantic Description of Register Transfer Language Elements and Mechanized Simulator Construction," *Proc. 4th Int'l. Symposium on Computer Hardware Description Languages*, Palo Alto, California, October 1979, pp. 69-74.

- Nagle, A. and Parker, A., "Algorithms for Multiple-Criterion Design of Micro-programmed Control Hardware," to be published in *Proc. 18th Annual Design Automation Conference*, Nashville, Tennessee, June 1981.
- Oakley, J., "Symbolic Execution of Formal Machine Descriptions," Ph.D. Thesis, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, April 1979.
- Parker, A.C., "Digital Interface Description," *Proc. COMPCON*, February 1978, pp.
- Parker, A., Thomas, D., Crocker, S., and Cattell, R., "ISPS: A Retrospective View," *Proc. 4th Int'l. Symposium on Computer Hardware Description Languages*, Palo Alto, California, October 1979, pp. 21-27.
- Parker, A. and Wallace, J., "SLIDE" An I/O Hardware Descriptive Language," to appear in *IEEE Trans. on Computers*, 1981.
- Patterson, D., "STRUM: Structured Microprogram Development System for Correct Firmware," *IEEE Trans. on Computers*, vol. C-25, no. 10, October 1976, pp. 974-985.
- Razouk, R., "The Graph Model of Behavior Simulator," *Proc. Symposium on Design Automation and Microprocessors*, 1977, pp. 67-76.
- Razouk, R. and Estrin, G., "Validation of the X.21 Interface Specification," *Trends and Applications*, May 1980, pp. 155-167.
- Roth, J.P., "Hardware Verification," *IEEE Trans. on Computers*, vol. C-26, no. 12, December 1977, pp. 1292-1293.
- Russell, R., "The PDP-11: A Case Study of How NOT to Design Condition Codes," *Proc. 5th Annual Symposium on Computer Architecture*, Palo Alto, California, April 1978, pp. 190-194.
- Sastry, S., information discussions, University of Southern California, 1980.
- Shiva, S.G., "Computer Hardware Description Languages - A Tutorial," *Proc. IEEE*, vol. 68, no. 12, December 1979, pp. 1605-1615.
- Snow, E., Siewiorek, D., and Thomas, D., "A Technology-Relative Computer Aided Design System: Abstract Representations, Transformations, and Design Tradeoffs," *Proc. IEEE Design Automation Conference*, June 1978, pp. 220-226.
- Sorensen, I.H., "System Modeling," Master's Thesis, Computer Science Department, University of Aarhus, Denmark, March 1978.
- Sunshine, C., "Formal Techniques for Protocol Specification and Verification," *Computer*, vol. 12, no. 9, September 1979.

- Thomas, D.E. and Siewiorek, D.P., "Measuring Designer Performance to Verify Design Automation Systems," *IEEE Trans. on Computers*, vol. C-30, no. 1, January 1981, pp. 48-60.
- Tredennick, T., "How to do Flowcharts for a Controller," IBM Research Report, RC 8426 (#36569), IBM - T.J. Watson Research Center, Yorktown Heights, New York, 1980.
- van-Mierop, D., "Design and Verification of Distributed Interacting Processes," Tech. Report UCLA-ENG-7920, Computer Science Department, University of California at Los Angeles, March 1979.
- van-Mierop, D., Crocker, S., and Marcus, L., "Verification of the FTSC Micro-program," *Proc. 11th Annual Microprogramming Workshop*, Pacific Grove, California, November 1978.
- Visser, C., "Interface, A Dispersed Architecture," *Proc. 3rd Annual Symposium on Computer Architecture*, 1976, pp. 98-104.
- Visser, C., "Formal Description with Asynchronous State Machines," unpublished manuscript, Twente University of Technology, Enschede, The Netherlands, November 1978.
- Wagner, T.J., "Hardware Verification," Ph.D. Thesis, Computer Science Department, Stanford University, Stanford, California, September 1977.
- Wallace, J., "On Automatic Verification of SLIDE Descriptions," Master's Thesis, Department of Electrical Engineering, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1979.
- Wallace, J. and Parker, A., "SLIDE" An I/O Hardware Descriptive Language," *Proc. 4th Int'l. Symposium on Hardware Descriptive Languages*, Palo Alto, California, October 1979, pp. 82-88.
- Winograd, T., "Beyond Programming Languages," *Comm. ACM*, vol. 22, no. 7, July 1979, pp. 391-401.
- Zimmerman, G., "The MIMOLA Design System: A Computer Aided Digital Processor Design Method," *Proc. 15th Design Automation Conference*, June 1979, pp. 53-58.

END

DATE
FILMED

6-83

DTIC